Enumeration and Deduction Driven Co-Synthesis of CCSL Specifications Using Reinforcement Learning

Ming Hu, Jiepin Ding, Min Zhang, Frédéric Mallet, Mingsong Chen

Enumeration and Deduction Driven Co-Synthesis of CCSL Specifications Using Reinforcement Learning

Ming Hu[†], Jiepin Ding[†], Min Zhang[†], Frédéric Mallet[‡], and Mingsong Chen[†]
[†]Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai 200062, China
[‡]Université Cote d'Azur, CNRS, Inria, I3S, France

***Abstract*—The Clock Constraint Specification Language (CCSL) has become popular for modeling and analyzing timing behaviors of real-time embedded systems. However, it is difficult for requirement engineers to accurately figure out CCSL specifications from natural language-based requirement descriptions. This is mainly because: i) most requirement engineers lack expertise in formal modeling; and ii) few existing tools can be used to facilitate the generation of CCSL specifications. To address these issues, this paper presents a novel approach that combines the merits of both Reinforcement Learning (RL) and deductive techniques in logical reasoning for efficient co-synthesis of CCSL specifications. Specifically, our method leverages RL to enumerate all the feasible solutions to fill the holes of incomplete specifications and deductive techniques to judge the quality of each trial. Our proposed deductive mechanisms are useful for not only pruning enumeration space, but also guiding the enumeration process to reach an optimal solution quickly. Comprehensive experimental results on both well-known benchmarks and complex industrial examples demonstrate the performance and scalability of our method. Compared with the state-of-the-art, our approach can drastically reduce the synthesis time by several orders of magnitude while the accuracy of synthesis can be guaranteed.**

***Index Terms*—Specification synthesis, reinforcement learning, logical clocks, deduction, enumeration**

I. Introduction

Model-driven design and development of real-time embedded systems usually adopt a top-down design flow [1], [2], where requirement engineers are required to derive formal specifications manually from textual design descriptions [3]. This may lead to serious problems, since a majority of requirement engineers have limited knowledge about formal methods. Worse still, due to the skyrocketing complexity of real-time embedded systems, it is hard for requirement engineers to figure out all the timing traces of a system at the design phase [4]. Consequently, most formal specifications are constructed simply based on limited observation of system timing traces, which makes the process of composing formal specifications more time-consuming and error-prone [5], [6].

The above dilemma also happens when adopting MARTE to design and implement embedded systems [7], [8], [9], [10], where MARTE is a promising UML profile for *Modeling and Analysis of Real-Time and Embedded systems*. In MARTE, there is a companion language called *Clock Constraint Specification Language* (CCSL) [11], [12], [13], which is devised to specify timing aspects of real-time systems. As a promising timing specification that is attracting more and more attention from both industrial and academic communities, CCSL treats logical clocks as first-class citizens to represent and constrain repetitive system events. CCSL gives a concrete syntax to handle Leslie Lamport's logical clocks [14] , while providing powerful extensions as introduced by synchronous languages [15]. It provides a comprehensive set of predefined clock constraints to capture classical causal and temporal relations among events, such as precedence, exclusion, and delay. Logical clock is a kind of abstraction of physical time. It captures logical relations of system events, and abstracts away the physical nature of time to avoid over-specification in the early phases allowing specifications to be modeled and reshaped as required [16]. Thus, CCSL is often used as a high-level formalism in the *Formal Specification Level* [17] to accurately model the causal and temporal timing behaviors of real-time embedded systems [18], [19], [20].

There exist various validation- and verification-based methods and tools that can effectively detect errors or check specific properties of CCSL specifications. They transform CCSL specifications into other existing formalisms such as automata [21], transition systems [22], [23], and SMT constraints [24]) and resort to off-the-shelf tools for verification and validation. Many case studies have shown the effectiveness and efficiency of these tools to detect potential flaws in the requirements and designs of embedded systems. However, one premise of the above approaches is that CCSL specifications must be completely ready-made. This is difficult to achieve in practice, since it is hard for requirement engineers to formally model timing constraints due to the lack of expertise and design automation tools.

Specification synthesis has proved to be an effective approach for developing formal specifications from scratch [25], [26]. It is assumed that requirement engineers can figure out partial specifications (i.e., sketches) including causal and temporal constraints (i.e., relations and expressions) among logical clocks, where the unknown parts of constraints are modeled as holes. As a first attempt for the synthesis of CCSL specifications, *CCSLSketch* [27] encodes an incomplete CCSL specification together with its expected timing traces (i.e., samples) as a sketching problem. Under the guidance of given timing traces, *CCSLSketch* can automatically fill the holes of incomplete constraints in the sketch based on a SAT solver. Nevertheless, since sketching is based on SAT solving, the scalability of *CCSLSketch* is strongly limited by

This paper has passed an Artifact Evaluation process.

the size of CCSL problems. Without taking the high-level syntax information of sketches into account, the underlying SAT solving of *CCSLSketch* can easily get stuck at local search, resulting in intolerable synthesis time. Note that essentially the CCSL synthesis process tries to explore all the hole combinations to find one optimal solution for filling holes. However, due to a large number of correlated clocks and operators, the enumeration space of CCSL synthesis is extremely large. Therefore, *how to efficiently figure out one best hole combination for a given partial CCSL specification satisfying all the given timing traces is becoming a major challenge in CCSL specification synthesis.*

Inspired by the studies introduced in [28], [29] that employ deduction to accelerate program synthesis, in this paper we propose a novel and efficient CCSL co-synthesis method that tightly couples the merits of enumeration and deduction. In our approach, we formulate the syntax-guided synthesis based on enumeration as a Reinforcement Learning (RL) problem. Starting from a blank model (i.e., policy) without any prior knowledge, our approach updates the policy on-the-fly during the synthesis and gradually improves the policy by incorporating the guidance made by a deduction engine. Specifically, our approach considers partial CCSL specifications as states in a Markov Decision Process (MDP) and filling holes as actions. In this way, a policy specifies how to fill the holes in an incomplete CCSL specification to form a specific and complete one. Our method consists of three major components, i.e., *Action*, *Deduce*, and *Update*. Given a partial CCSL specification *Spec* and its current policy *P*, *Action* fills one hole of the *Spec* to get a more complete specification *Spec′*. Then *Deduce* uses our proposed deductive reasoning methods to check whether the *Spec′* is feasible. If yes, *P* will be *Updated* with a positive reward to improve *P*. Otherwise, *P* will be *Updated* with some penalty (negative reward). Once *P* converges, it can derive an optimal synthesis solution for the incomplete CCSL specification. Within the whole synthesis process, based on both specific characteristics of CCSL and collected timing traces, our approach adopts a powerful deduction engine that can check whether a filled CCSL specification is feasible at its earliest stage. The goal of our RL-based method is to improve the policy over time under the help of this deduction engine. This paper makes the following **three contributions**:

- Based on our proposed state representation and reward design, we present a novel RL-based co-synthesis framework that leverages the synergy between enumeration and deduction for the automated generation of CCSL specifications.
- We introduce various deduction reasoning mechanisms to facilitate the CCSL synthesis process, which can not only enhance the capability of enumeration space pruning, but also accelerate the search for optimal solutions.
- We conduct comprehensive experiments on both well-known benchmarks from *CCSLSketch* [30] and complex industrial examples. Experimental results show that, compared with state-of-the-art, our approach can significantly reduce the synthesis time with the accuracy guaranteed.

Paper organization: Section II introduces the preliminaries and notations used in RL-based CCSL synthesis. Section III presents the details of our enumeration and deduction-driven CCSL synthesis method. Section IV presents the experimental results. Section VI presents the related work of program synthesis and CCSL specification formalization. Finally, Section VI concludes the paper.

II. Preliminaries

A. Clock Constraint Specification Language (CCSL)

CCSL is a domain-specific specification language, proposed for modeling timing behaviors in real-time and embedded systems. One key feature of CCSL is that it makes use of logical clocks to represent timing constraints on events. Logical clocks can be considered as an abstraction of physical clocks by ignoring the exact time when events and behaviors occur but only capturing the logical order among them instead. With logical clocks, partial ordering of events can be obtained without recourse to any physical "real" time [31].

Definition 2.1 (Logical Clocks): A logical clock c is a predicate over natural numbers. For any $i \in \mathbb{N}^+$, $c(i)$ means that c ticks at step i when c is true at the same step. Otherwise, c idles at step i. ■

In CCSL there are two types of clocks, i.e., atomic clocks and expression clocks. An atomic clock represents a physical event in systems, while an expression clock represents an event that is composed of existing clocks with a certain relation. Relations of clocks are defined by two types of operators: i) binary operators (i.e., relation operators), and ii) definitional operators (i.e., expression operators). We denote them by $O_b = \{=, \prec, \preceq, \subseteq, \#\}$ and $O_d = \{+, *, \wedge, \vee\}$, respectively. The operators in O_b define binary relations between two clocks. Those in O_d compose a new clock with existing clocks. Table I presents the syntax of all the constraints and their meanings.

Definition 2.2 (CCSL Constraints): A CCSL constraint is in one of the following forms:

$$c_1 \ ? \ c_2, \text{ where } ? \in O_b \tag{1}$$

$$c_e \ \triangleq c_1 \ ? \ c_2, \text{ where } ? \in O_d \tag{2}$$

$$c_e \ \triangleq c \ \$ \ d, \text{ where } d \in \mathbb{N}^+ \tag{3}$$

$$c_e \ \triangleq c \ \propto \ p, \text{ where } p \in \mathbb{N}^+ \tag{4}$$

■

The semantics of CCSL operators can be formally defined in terms of *schedule* and *history*. A schedule determines the behavior of logical clocks at any step. It is formalized as a total function $\delta : \mathbb{N}^+ \to 2^C$ for a set C of clocks, where 2^C denotes the power set of C. For each step $n \in \mathbb{N}^+$, a clock $c \in C$ is in $\delta(n)$ if and only if c ticks at step n. Given a schedule δ and clock c, we use $\chi_\delta(c,n)$ to denote the number of ticks that c ticks before step n. χ_δ is called the *history* of δ.

Table I presents the syntax of CCSL operators, whose semantics are defined by the satisfaction of a schedule against corresponding constraints. For instance, given a schedule δ and a constraint $c_1 \prec c_2$, we say δ satisfies $c_1 \prec c_2$, denoted

TABLE I
Syntax of CCSL operators and their semantics

Name	**Constraint**	**Semantics** ($\delta \models \phi$)	**Description**
Coincidence	$c_1 = c_2$	$\chi_\delta(c_1,i) = \chi_\delta(c_2,i)$	c_1 always occurs equal to c_2.
Precedence	$c_1 \prec c_2$	$(\chi_\delta(c_1,i) = \chi_\delta(c_2,i)) \implies c_2 \notin \delta(i)$	c_1 always occurs strictly more than c_2.
Causality	$c_1 \preccurlyeq c_2$	$\chi_\delta(c_1,i) \geq \chi_\delta(c_2,i)$	c_1 always occurs greater than or equal to c_2.
Subclock	$c_1 \subseteq c_2$	$c_1 \in \delta(i) \implies c_2 \in \delta(i)$	whenever c_1 occurs, it cause c_2 to occur.
Exclusion	$c_1 \# c_2$	$c_1 \notin \delta(i) \vee c_2 \notin \delta(i)$	c_1 and c_2 will never occur simultaneously.
Union	$c_1 \triangleq c_2 + c_3$	$c_1 \in \delta(i) \iff c_2 \in \delta(i) \vee c_3 \in \delta(i)$	Whenever either c_2 or c_3 ticks, c_1 ticks.
Intersection	$c_1 \triangleq c_2 * c_3$	$c_1 \in \delta(i) \iff c_2 \in \delta(i) \wedge c_3 \in \delta(i)$	Whenever both c_2 and c_3 ticks, c_1 ticks.
Infimum	$c_1 \triangleq c_2 \wedge c_3$	$\chi_\delta(c_1,i) = max(\chi_\delta(c_2,i), \chi_\delta(c_3,i))$	c_1 is the fastest clock but slower than c_2, c_3.
Supremum	$c_1 \triangleq c_2 \vee c_3$	$\chi_\delta(c_1,i) = min(\chi_\delta(c_2,i), \chi_\delta(c_3,i))$	c_1 is the slowest clock but faster than c_2, c_3.
Delay	$c_1 \triangleq c_2 \$ d$	$\chi_\delta(c_1,i) = max(\chi_\delta(c_2,i) - d, 0)$	c_1 is delayed by c_2 with d time units.
Periodicity	$c_1 \triangleq c_2 \propto p$	$c_1 \in \delta(i) \iff c_2 \in \delta(i) \wedge \exists j \in \mathbb{N}^+ . \chi_\delta(c_2,i) = j \times p - 1$	c_1 periodically ticks with c_2 every p times.

by $\delta \models c_1 \prec c_2$, if and only if for all $i \in \mathbb{N}^+$ the condition $(\chi_\delta(c_1,i) = \chi_\delta(c_2,i)) \implies c_2 \notin \delta(i)$ holds. The condition specifies that, whenever c_1 and c_2 have ticked the same number of ticks before step n, c_2 should never occur at step n. In other words, c_1 always ticks strictly more than c_2. Consequently, we can conclude that c_1 precedes c_2. Precedence is an important constraint type. For instance, $put \prec fetch$ describes that the *fetch* operation on buffers must always follow the *put* operation. In other words, *put* must always occur strictly more than *fetch*. Due to limited space, we do not provide full details of the formal semantics of other CCSL constraints. Please refer to [13] for more details.

Theoretically, a set of CCSL constraints define all possible schedules that conform to all the constraints in the set. Thus, schedulability is one of the fundamental properties that should be satisfied [32]. If a set of constraints are not schedulable, it means that there must be conflicts in the constraints and the conflicts would result in a deadlock of systems. For instance, if there exist two constraints $put \prec fetch$ and $fetch \prec put$ in a set, both the corresponding events of the two clocks cannot occur because they precedes each other, which is apparently impossible. Such conflicts can be easily detected in CCSL using SMT-based or automata-based approaches in the design but not implementation phase [22], [32]. Besides scheduling analysis, CCSL constraints can be used to check whether they satisfy more expressive temporal properties such as those defined by Linear Temporal Logic (LTL) formulas [24], and to monitor system execution at run time [33]. In practice, schedules are partially reflected by system traces. A *trace* is a finite sequence of system behaviors logged by its occurring time. We say that constraints admit an execution trace if the trace conforms to the constraints.

B. CCSL Specification Synthesis

At the very initial phase of requirement design, it is common to have many uncertainties. As a result, requirement engineers cannot write a full specification with all information fixed. Typically, they leave these uncertainties as blanks in constraints, leading to incomplete specifications. To formalize incomplete CCSL specifications, we introduce four types of holes: i) $\square_o^b$ denotes a relation operator hole; ii) $\square_o^d$ denotes an expression operator hole; iii) $\square_c^b$ indicates a relation clock hole; and iv) $\square_c^d$ indicates an expression clock hole. Each hole is a placeholder for its corresponding clocks and operators, e.g., relation operator hole $\square_o^b$ and expression operator hole $\square_o^d$ for the operators in O_b and O_d, respectively.

*Definition 2.3 (*CCSL *Constraint with Hole):* A constraint with a hole is a CCSL constraint that contains one and only one hole for either a clock or an operator. ■

A set of incomplete CCSL constraints is the union of a set of regular CCSL constraints and a set of constraints with holes. The synthesis of CCSL specifications is a problem of automatically generating full CCSL specifications from incomplete ones according to additional information such as execution traces of systems. An execution trace is an ordered sequence of events that occur temporally. Synthesized specifications should admit all the provided execution traces.

Definition 2.4 (CCSL Specification Synthesis Problem): Given a CCSL specification Φ with incomplete constraints and a set T of expected system traces, the synthesis problem is to fill all the holes in the constraints such that the completed Φ admits all the traces in T. ■

There can be more than one solution of filling the holes. We say a solution s is tighter than another s' if s covers fewer traces that are not in T than s'. An optimal solution covers only the exact traces in T. It is possible that there are no candidate clocks and operators, i.e., there are some semantic conflicts among the constraints. Detecting and fixing such conflicts earlier before system implementation is desirable from the point of development view.

Definition 2.5 (Specification Tightness): Let Φ and Φ' be two completed versions of an incomplete CCSL specification $\Phi_\square$. For any schedule δ, if $\delta \models \Phi$ implies $\delta \models \Phi'$, then we say Φ is tighter than Φ'. If there is no completed specification tighter than Φ, then Φ is the tightest specification for $\Phi_\square$. ■

Since our approach adopts RL for synthesis, the generated solutions at different time may be different due to the randomness. To enable fair comparison, in the experiment section we repeat the RL-based synthesis process many times. Based on all the generated solutions, we calculate the percentage of the

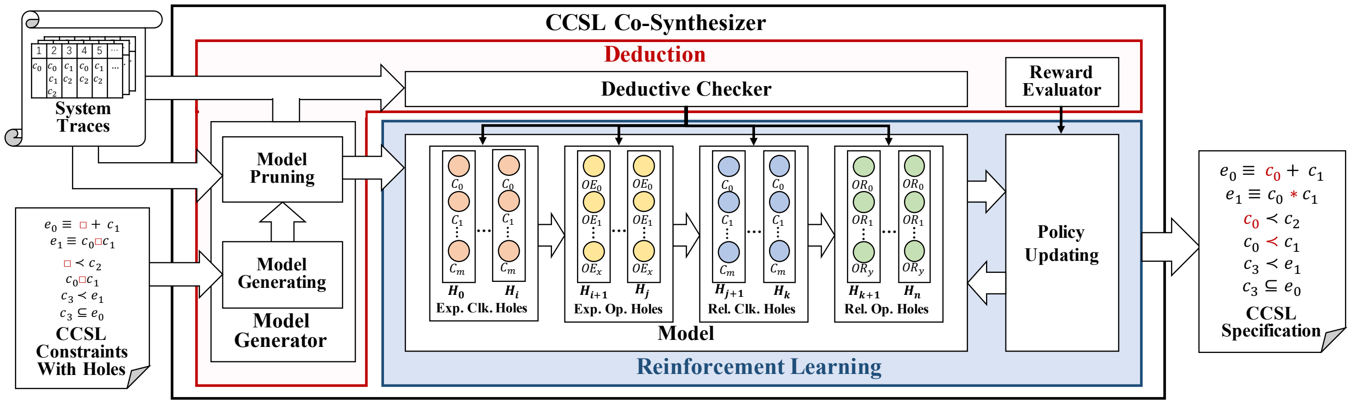


Fig. 1. Workflow of our CCSL co-synthesizer

tightest solutions to indicate the accuracy of synthesis. We call the tightest solutions *golden references* when they are clear in the context. Note that the definition of synthesis accuracy here is different from the one proposed in [27], which is calculated based on the similarity between the bounded schedules generated by the synthesized specification and the bounded schedules derived from its golden reference counterpart.

Definition 2.6 (Synthesis Accuracy): Let $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ be a finite set of synthesized solutions for a given incomplete CCSL specification Φ and the set of expected system traces T. Let $\Sigma' = \{\sigma'_1, \sigma'_2, \ldots, \sigma'_m\}$ be a subset of Σ that contains all the tightest solutions in Σ. The synthesis accuracy equals to the percentage of the tightest solutions in Σ, i.e., $\frac{m}{n}$. ■

C. Reinforcement Learning

Along with the prosperity of Artificial Intelligence (AI), we are witnessing that deep learning techniques can not only deal with traditional tasks such as classification and natural language processing, but also be used to address logical reasoning problems, such as program synthesis [28], [34] and satisfiability problem solving [35], [36]. As a promising decision making approach, Reinforcement Learning (RL) is more appropriate than other machine learning methods, since the goal of such RL problems is to determine concrete values for unknown variables in problems [37].

An RL model indicates a Markov Decision Process (MDP) [38]. It can be represented as a 4-tuple $\langle S, A, F, R\rangle$, where:

- S is a set of states,
- A is a set of actions,
- $F : S \times A \to S$ is a transition function from states to their successors caused by actions,
- $R : S \to \mathbb{R}$ is a reward function.

In RL, agents get rewarded or punished according to their decisions, judged by R. The goal of RL is to learn a policy π to reach a terminal state and get the maximum reward. Q-learning [38] is a classical RL algorithm based on a Q-Table to record the credits that agents can get from decision-making. For a given Q-Table, we use $Q(s,a)$ to record the expectation of the maximum reward for taking action $a \in A(s)$ in state $s \in S$. In the training process of Q-learning, there are multiple rounds of search. In each round, the Q-Table will be updated using the Bellman equation defined as follows:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a)]. \quad (5)$$

In the equation, α indicates the learning rate, $\max_{a'} Q(s',a')$ represents the maximum reward expectation for the next state, and γ is the discount rate. The equation makes the value of $Q(s,a)$ approach to the maximum reward expectation through iterative learning.

III. RL-Based CCSL Co-Synthesis

The basic idea of our approach is to transform the CCSL specification synthesis problem into an RL problem. Unlike traditional RL tasks, we introduce a deductive algorithm to guide the whole learning process. The purpose of the deductive algorithm is to prune unnecessary trials under the latest decision. Trials are unnecessary when they are checked logically invalid by deductive approaches.

Figure 1 depicts an overview of our enumerative and deductive co-synthesis approach. The approach takes two inputs, i.e., system traces and a set of incomplete CCSL constraints and the output is a complete CCSL specification that satisfies the given system traces. Our approach consists of two parts, i.e., RL and deduction. The RL part explores the solution space of given incomplete CCSL constraints to find the golden solution and the deduction part has three main functions: i) generating and pruning the initial model, ii) verifying whether the solution of RL exploration satisfies the given system traces, and iii) evaluating the reward of the solution. The workflow of our approach is as follows. Firstly, an initial RL model is constructed according to the input constraints. Secondly, the initial model is refined to prune unnecessary nodes and corresponding connections according to the input system traces. Finally, the model is trained using RL until all the constraints are completed. The following subsections will detail the key step of our approach.

A. Encoding CCSL Synthesis into an MDP

The synthesis of CCSL is regarded as a process of filling holes in constraints. The process can be modeled as an MDP, where agents get a reward or punishment according to their decisions of choosing which clocks or operators to fill the holes. Essentially, a state in the MDP is a list of holes. An action is a behavior of filling one hole in the list. The action causes a new list, which is considered as a transition in MDP. More precisely, we formulate the synthesis of an incomplete CCSL specification as an MDP $M = (S, \mathcal{A}, \mathcal{F}, \mathcal{R})$, where:

- S is a set of lists whose element is a 2-tuple $\langle \square^i, v \rangle$, where $\square^i$ is an indexed hole in the given specification and v is a candidate value of $\square^i$. The element indicates that hole $\square^i$ has been filled by v. Note that v equals *null* means that $\square^i$ has not been filled. Index i indicates the position of the hole in the list. For an incomplete CCSL specification, its initial list indicates that no hole is filled, i.e., the candidate value of each element is *null*. On the contrary, the termination list indicates that all holes are filled, i.e., there is no element with *null* candidate in the list. Note that to avoid duplicate searches, we require that holes must be filled in order. For elements $\langle \square^i, v_i \rangle$ and $\langle \square^j, v_j \rangle$ in one list, if $i < j$ and $v_i = null$, then $v_j = null$.
- $\mathcal{A}$ is a set of mappings $(\square^i, v)$ from an indexed hole $\square^i$ to a candidate value v, which denotes the action using v to fill $\square^i$. A value can be a clock or an operator, depending on the hole type.
- $\mathcal{F}$ is a set of transitions. Each transition is in the form of $\alpha \xrightarrow{(\square^i, v)} \alpha'$, where α' is a list that is obtained by replacing the hole $\square$ at index i in α with the value v.
- $\mathcal{R}$ is a reward function that is defined in the form of $\mathcal{R}(s) = RewardEvaluator(s)$. The reward evaluator $RewardEvaluator$ is predefined function to reward or punish the system when it reaches state s.

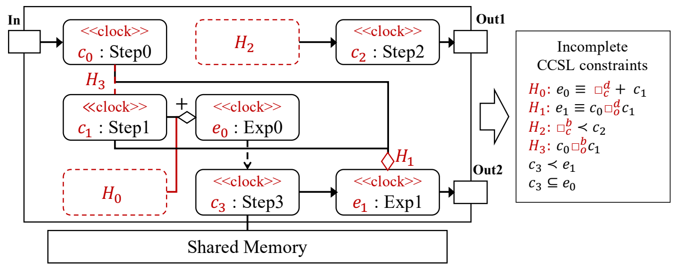


Fig. 2. An example of incomplete CCSL specification generation

Figure 2 presents an example to illustrate how to generate an incomplete CCSL specification. The left part of the figure gives the design of an application in the form of a UML structured class. This application captures one input *In*, performs six calculations (i.e., *Step0, Step1, Step2, Step3, Exp0*, and *Exp1*), and produces two results, i.e., *Out1* and *Out2*. Here, we use rounded rectangles to indicate the calculation. The solid arrow line and dashed arrow line indicate the precedence and inclusion relations between calculations, respectively. The diamond notation denotes the aggregation relation. At the beginning of the design, we assume that there are four uncertain components in the application, which are denoted by H_0, H_1, H_2, and H_3. Note that H_0 and H_2 are placeholders for the six calculations. H_1 is an expression placeholder indicating that the calculation *Exp1* is based on some unknown combination of *Step0* and *Step1*. H_3 is a relation placeholder to denote the unknown relationship between *Step0* and *Step1*. Especially, H_0 indicates an unknown calculation that should follow the condition, i.e., if *Exp0* is executed, then either H_0 or *Step1* should be executed at the same time.

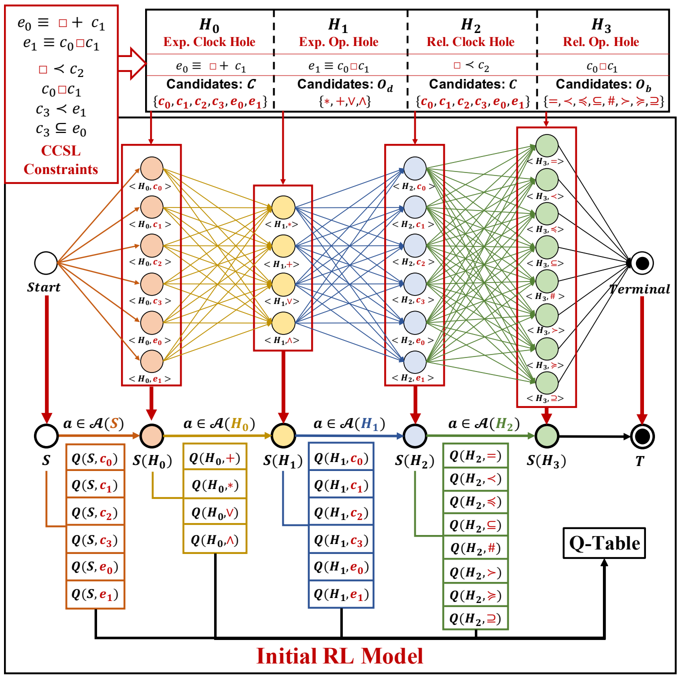
To formalize the above design in CCSL, we need to create one clock for each component in the class diagram. The right part of Figure 2 shows the obtained incomplete CCSL specification with nine clocks including 4 atomic clocks (i.e., c_0, c_1, c_2, and c_3) and 2 expression clocks (i.e., e_0 and e_1). The specification has six constraints with four holes. During the synthesis, we can figure out the candidates for each hole based on its type. For example, H_0 is an expression clock hole and H_2 is a relation clock hole. According to their hole types, H_0 and H_1 can select their candidates from the clock set $C = \{c_0, c_1, c_2, c_3, e_0, e_1\}$. Since H_1 is an expression operator hole, it can select candidates from the set $O_d = \{+, *, \wedge, \vee\}$. H_3 is a relation operator hole, thus it can select candidates from $O_b = \{=, \prec, \preceq, \subseteq, \#, \succ, \succeq, \supseteq\}$. Based on the above information, we can construct an MDP $M = (S, \mathcal{A}, \mathcal{F}, \mathcal{R})$ for the incomplete CCSL specification as follows.

- **State set** S. Since there are four holes in the specification, an MDP state is a list in the form of $[\langle H_0, v_0 \rangle, \langle H_1, v_1 \rangle, \langle H_2, v_2 \rangle, \langle H_3, v_3 \rangle]$ where v_i is a candidate value of H_i or *null*. Note that we fill the holes one by one along with the list. In other words, we cannot fill H_{i+1} until H_i has been filled. If the value of the i^{th} element is *null*, the values of following holes H_j $(j > i)$ in the list should be also *null*. The initial MDP state is $[\langle H_0, null \rangle, \langle H_1, null \rangle, \langle H_2, null \rangle, \langle H_3, null \rangle]$, and the MDP has $|C| \times |C| \times |O_t| \times |O_b| = 1152$ terminal states. For example, $[\langle H_0, c_0 \rangle, \langle H_1, * \rangle, \langle H_2, c_0 \rangle, \langle H_3, + \rangle]$ is a terminal state.
- **Action set** $\mathcal{A}$. $\mathcal{A}$ is a set of mappings in the form of (H_i, v_i), where v_i is a candidate value of H_i. For example, the action set of H_0 is $\mathcal{A}(H_0) = \{(H_0, c_0), (H_0, c_1), (H_0, c_2), (H_0, c_3), (H_0, e_0), (H_0, e_1)\}$. For the MDP, $\mathcal{A}$ equals to the union of the action sets of the four holes.
- **Transition set** $\mathcal{F}$. Each element of $\mathcal{F}$ is in the form of $\alpha \xrightarrow{(H_i, v_i)} \alpha'$ where α is a non-terminal state and α' is a non-initial state in S. The transition can be enabled only when the i^{th} element of the current state is *null* and the $(i-1)^{th}$ element of the current state is not *null*. When the transition is triggered, the i^{th} element of the current state (i.e., α) is filled with v_i.
- **Reward function** $\mathcal{R}$. Please refer to Section III-C3 for more details.

B. Initial RL Model Generation and Pruning

According to the definition of MDP, we implement a model generator to construct the RL model for CCSL co-synthesis. The generator consists of two parts, i.e., initial RL model generation and model pruning. The former is used to automatically generate an RL model from an incomplete CCSL specification, and the latter is used to simplify the initial model according to the provided system timing traces.

1) Initial Model Generation: We construct a multi-layer network to implement an MDP. Figure 3 shows the process of model generation based on the example shown in Figure 2. Firstly, holes are sorted and indexed according to their types. Each hole has a finite set of candidate values. We use a node to represent the result after filling some hole with a candidate value. All the nodes of the hole construct a network layer. For instance, the hole H_0 corresponds to incomplete constraint $e_0 \equiv \Box_c^d + c_1$ in Figure 3. There are six candidate clocks assigned to the hole. Therefore, we construct six nodes, each of which represents one possible assignment. The six nodes form the first network layer, while the other layers are constructed in the same manner. A layer is fully connected to its successor layer, representing all possible combinations. We add two extra states, i.e., *Start* and *Terminal*, to represent the start and termination of the synthesis. A valid path from *Start* to *Terminal* indicates a feasible solution to fill all the holes, thus a complete specification is generated.



$e_0 \equiv \Box + c_1$
$e_1 \equiv c_0 \Box c_1$
$\Box \prec c_2$
$c_0 \Box c_1$
$c_3 \prec e_1$
$c_3 \subseteq e_0$
CCSL Constraints
H_0 Exp. Clock Hole
H_1 Exp. Op. Hole
H_2 Rel. Clock Hole
H_3 Rel. Op. Hole
Candidates: C $\{c_0, c_1, c_2, c_3, e_0, e_1\}$
Candidates: O_d $\{*, +, \vee, \wedge\}$
Candidates: O_b $\{=, \prec, \preccurlyeq, \subseteq, \#, \succ, \succcurlyeq, \supseteq\}$
Start
Terminal
$a \in \mathcal{A}(S)$
$a \in \mathcal{A}(H_0)$
$a \in \mathcal{A}(H_1)$
$a \in \mathcal{A}(H_2)$
S
$S(H_0)$
$S(H_1)$
$S(H_2)$
$S(H_3)$
T
$Q(S,c_0)$ $Q(S,c_1)$ $Q(S,c_2)$ $Q(S,c_3)$ $Q(S,e_0)$ $Q(S,e_1)$
$Q(H_0,+)$ $Q(H_0,*)$ $Q(H_0,\vee)$ $Q(H_0,\wedge)$
$Q(H_1,c_0)$ $Q(H_1,c_1)$ $Q(H_1,c_2)$ $Q(H_1,c_3)$ $Q(H_1,e_0)$ $Q(H_1,e_1)$
$Q(H_2,=)$ $Q(H_2,\prec)$ $Q(H_2,\preccurlyeq)$ $Q(H_2,\subseteq)$ $Q(H_2,\#)$ $Q(H_2,\succ)$ $Q(H_2,\succcurlyeq)$ $Q(H_2,\supseteq)$
Q-Table
Initial RL Model

Fig. 3. Initial RL model generation

In RL, the reward mechanism is used to measure the quality of decision-making. The measuring result is represented by a Q-Table, which records the expectation of the sum of rewards for each candidate to fill in the next hole. In our approach, since the number of states increases exponentially with the number of holes, it is unwise to use the traditional Q-Table structure, where each row of the Q-Table is a state (i.e., a list of 2-tuples in the form of $\langle \Box^i, v \rangle$) indicating a path from the *Start* node. In Figure 3, if we use the traditional RL method, the Q-Table will have $2+(6+6\times4+6\times4\times6+6\times4\times6\times8) = 1328$ rows. Unlike traditional RL models, in our approach nodes in the same layer share the same row of a Q-Table, indicating a filling operation for a specific hole. Therefore, our approach can drastically reduce the size of Q-Tables.

To guarantee the convergence of learning with the shrunk Q-Table, we introduce a dynamic learning rate to augment reward and penalty effects according to the judgment result. Since our goal is to find a strategy that maximizes the sum of rewards, the search with the highest sum of rewards should be first to learn. Based on this idea, we record the highest reward strategy in the search history. When the sum of rewards of a new search is greater than or equal to the highest one in history, we give a higher learning rate to the search. Otherwise, we give a lower learning rate.

2) RL Model Pruning: For an initial RL model, there may exist some connections and nodes that cannot admit the provided traces. In this case, such connections and nodes can be safely pruned without affecting the final learning results. To achieve a compact initial RL model, we resort to the rules of the number of clock ticks as shown in Table II to enable the aforementioned pruning on useless connections and node. In this table, we use $len(c)$ to denote the number of ticks for clock c within a given trace.

TABLE II
RULES OF THE NUMBER OF CLOCK TICKS FOR CCSL CONSTRAINTS

CCSL Constraint	Rule
$e_0 \triangleq c_1 + c_2$	$max(len(c_1), len(c_2)) \leq len(e_0) \leq len(c_1) + len(c_2)$
$e_0 \triangleq c_1 * c_2$	$len(e_0) \leq min(len(c_1), len(c_2))$
$e_0 \triangleq c_1 \vee c_2$	$len(e_0) = max(len(c_1), len(c_2))$
$e_0 \triangleq c_1 \wedge c_2$	$len(e_0) = min(len(c_1), len(c_2))$
$c_0 = c_1$	$len(c_0) = len(c_1)$
$c_0 \prec c_1$	$len(c_0) \geq len(c_1)$
$c_0 \preccurlyeq c_1$	$len(c_0) \geq len(c_1)$
$c_0 \subseteq c_1$	$len(c_0) < len(c_1)$

As a preprocessing procedure, our pruning method tries to delete unfruitful nodes in RL models based on all the given traces. For each trace, firstly we compute the number of ticks for each clock. Then, for each node we check whether the corresponding constraint satisfies all the rules presented in Table II. If not, we can delete the node by setting the corresponding Q-Table cell with a value of -1. The whole pruning process finishes when all the traces are checked. Figure 4 shows an example of model pruning. Assume that we only have one trace with a length of 20 as shown on the top right of the figure. Let us take the node $\langle H_1, + \rangle$ into account, which tries to fill the second hole and get the constraint $e_1 \equiv c_0 + c_1$. According to the rules in Table II and the CCSL constraints shown on the top left of Figure 4, we need to check two constraints, i.e., $e_1 \equiv c_0 + c_1$ and $c_3 \prec e_1$. Based on the first and sixth rules in Table II, these two constraints contradict with each other, since $14 \leq len(e_1) \leq 27$ and $len(e_1) \leq 12$. Therefore, the node can be pruned from the Q-Table safely. Based on the given trace, 12 nodes are pruned as shown in Figure 4.

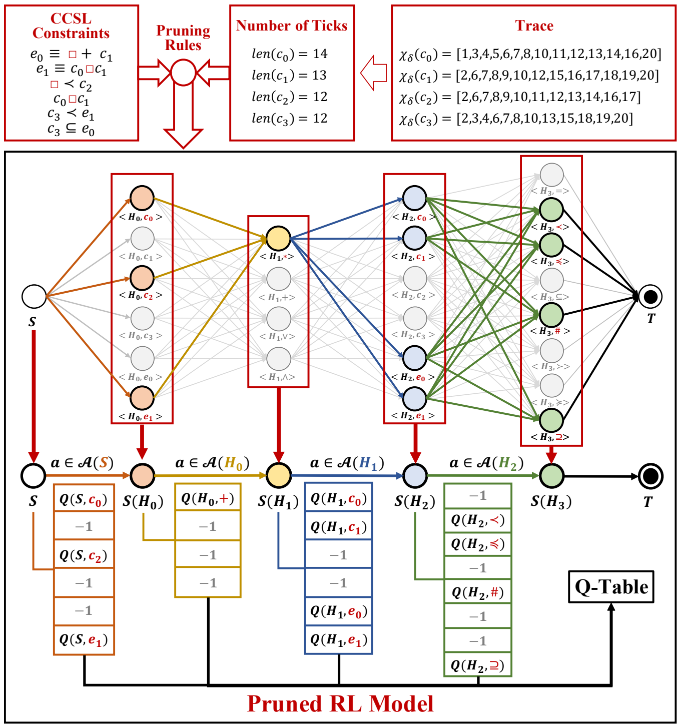


Fig. 4. RL model pruning

```
Algorithm 1: CCSL Specification Co-Synthesis
Input:
i) Φ, a CCSL specification with holes;
ii) T, a set of system timing traces;
iii) π_0, an initial policy;
iv) round, the number of training times.
Output:
i) Spec, a complete CCSL specification.
1  CCSLSynthesizer(Φ,T,π_0,round) begin
2      π_θ ← π_0, maxSumReward ← 0;
3      for i ← 1 to round do
4          S ← Φ;
5          ValueList ← {};
6          unActList ← {};
7          for H ∈ Spec.holes do
8              choose a ∼ π_θ(H) using ϵ-greedy strategy;
9              ValueList.append((H,a));
10             S ← Fill(S,(H,a));
11             (r,π_θ,unActList,T) ←
                 CCSLChecker(S,ValueList,π_θ,unActList,T);
12             if r == −1 then
13                 break;
14             end
15         end
16         if r == 1 ∧ unActList == {} then
17             (π_θ,maxSumReward) ←
                 PolicyUpdate(ValueList,S,π_θ,maxSumReward,i);
18         end
19     end
20     return ResultEvaluator(Φ,π_θ);
21 end
```

C. Deduction-guided Model Training

Different from traditional RL algorithms, we use deductive reasoning to guide the search and reward evaluation. Once a value is determined for a hole, we simply check whether the filling violates the provided traces. When all the holes are filled, we need to evaluate the reward of each filled value and update the Q-Table correspondingly.

1) The training algorithm: Algorithm 1 presents our co-synthesis algorithm in detail. Note that the initial policy π_0 (i.e., Q-Table) is a pruned model that is obtained in the model generation phase. In each RL round (lines 3-19), we select actions to fill the holes. Line 8 denotes the action selection using the ϵ-greedy strategy. Line 9 appends the selection result to *ValueList*, which stores the current hole filling status. Line 10 fills the hole H with the selected action a and updates the specification S. Line 11 checks whether the incrementally filled specification can admit all the provided traces in T, and saves the checking result in r. Here, the function *CCSLChecker* is defined in Algorithm 2. We use *unActList* to store uncertain selections, which cannot be determined by *CCSLChecker*. Please see Algorithm 2 for more details. In line 12, if r equals to -1, the algorithm will stop the current round and start a new round. When one round of hole filling finishes, if r equals to 1 and *unActList* is empty, lines 16-18 will update the policy using *PolicyUpdate* as defined in Algorithm 3. Finally, line 20 reports a complete specification for Φ based on the newly learned policy π_θ.

2) Deductive checking: As aforementioned, once a value is assigned to a hole, we need to deductively check whether the filling violates any provided traces. Algorithm 2 details our deductive checking method. In line 2, we get the latest selection. Line 3 uses *Deduce* to check whether the selection admits the given traces. The function *Deduce* has three return values, i.e., -1, 0 and 1, which indicate that the selection is invalid, unknown and valid, respectively. If $r == -1$, lines 5-11 will update the policy. Here, line 5 uses *checkDep* to check whether the filled hole is related to other holes. If *checkDep* returns *true*, lines 6-8 will update the policy by subtracting a small penalty value Γ from the Q value of each selection stored in *ValueList*. If *checkDep* returns *false*, line 10 will set the Q value for the latest selection to -1. If r equals to 0, it means that the validity of the latest selection is uncertain. As an example shown in Figure 3, assume that we make a latest selection $\langle H_0, e_1 \rangle$. Since the hole H_1 is not filled, it is impossible to figure out the timing behaviors for both e_0 and

Algorithm 2: Deductive Checking

```
Input:
i) $S$, an incomplete specification;
ii) $ValueList$, a list of filled holes and their selected actions;
iii) $\pi_\theta$, a policy;
iv) $T$, a set of system timing traces;
v) $unActList$, a list of uncertain holes.
Output:
i) $r$, checking result;
ii) $\pi_\theta$, an updated policy;
iii) $unActList$, an updated list;
iv) $T$, a set of updated traces.
1  CCSLChecker($S$,$ValueList$,$\pi_\theta$,$T$,$unActList$) begin
2      ($H$,$a$)$\leftarrow$ $ValueList.getLast()$;
3      $r \leftarrow Deduce((H,a),T)$;
4      if $r == -1$ then
5          if $checkDep((H,a),S)$ then
6              for $(H',a') \in ValueList$ do
7                  $\pi_\theta(H',a') \leftarrow \pi_\theta(H',a') - \Gamma$;
8              end
9          else
10             $\pi_\theta(H,a) \leftarrow -1$;
11         end
12     else if $r == 0$ then
13         $unActList.append(H,a)$;
14     else
15         $(r,\pi_\theta,unActList,T) \leftarrow UpdateUnAct(unActList,\pi_\theta,T,S)$;
16     end
17     return $(r,\pi_\theta,unActList,T)$;
18 end
```

e_1 at this stage. Consequently, the *Deduce* function cannot determine the validity of the selection. In this case, line 13 will append the uncertain selection to *unActList*. If *r* equals to 1, it means that the selection is valid. In this case, line 15 will check the validity of each selection in *unActList*. If an unknown selection in *unActList* becomes valid, it will be removed from *unActList*. If an unknown selection in *unActList* becomes invalid, *UpdateUnAct* will update the policy in the same way as the one shown in lines 5-11. The function *UpdateUnAct* iterates until all the elements in *unActList* become uncertain. Finally, line 15 returns all the checking results.

3) Deductive reward evaluation for policy updating: The reward mechanism plays an important role in our approach, since it faithfully determines the quality of synthesized specifications. Aiming to achieve the tightest solutions, we propose a deductive method to evaluate the enumerated solutions by analyzing the logical connections between CCSL constraints. Based on the obtained reward evaluation results, the policies will be updated. Algorithm 3 details our reward evaluation and policy updating process. In line 5, we use the reward function *rewardEvaluator* to evaluate each selection and calculate its reward. We define the reward in the range of $[0,1]$ and use different evaluation strategies on the four types of holes. Table III presents our reward settings for different kinds of holes. In this table, the first column denotes the hole types, the second column indicates the positions of holes, and the remaining five columns present the reward values for different hole types. Note that except for relation operation holes, the reward calculation of all the holes with other types needs to take the hole position into account.

Algorithm 3: Deductive Reward Evaluation

```
Input:
i) $S$, filled CCSL specification;
ii) $ValueList$, a list of filled values and their holes;
iii) $\pi_\theta$, policy;
iv) $maxSumReward$, maximum sum of rewards;
v) $i$, the $i^{th}$ round of training.
Output:
i) $\pi_\theta$, updated policy.
1  PolicyUpdate($S$,$ValueList$,$\pi_\theta$,$maxSumReward$,$i$) begin
2      $sumReward \leftarrow 0$, $rewardList \leftarrow \{\}$;
3      for $i$ in len(ValueList) to 1 do
4          $(H,a) \leftarrow ValueList[i]$;
5          $reward \leftarrow rewardEvaluator((H,a),S)$;
6          $rewardList.add(reward)$;
7          $sumReward \leftarrow sumReward + reward$;
8      end
9      $maxSumReward \leftarrow Max(maxSumReward, sumReward)$;
10     $\delta \leftarrow maxSumReward - sumReward$;
11     for $i$ in 1 to len(ValueList) do
12         $\pi_\theta(H,a) \leftarrow \pi_\theta(H,a) + (sumReward - \pi_\theta(H,a)) \times \alpha(\delta, round_i)$;
13         $sumReward \leftarrow sumReward - rewardList[i]$;
14     end
15     return $(\pi_\theta,maxSumReward)$;
16 end
```

For relation operator holes, we conduct reward evaluation based on the refinement sequence "$=,\subseteq,\prec,\preccurlyeq$" introduced by [27]. In the refinement sequence, an operator X is a refinement of another operator Y (denoted by $X \sqsupseteq Y$), if for any two clocks (i.e., c and c'), $(c,c') \in X$ implies $(c,c') \in Y$. The first row in Table III shows the reward settings for relation operator holes. Similar to the specification synthesis approach proposed in [27], we define the reward for relation operators based on their order in the refinement sequence. We set "=" with the highest reward (i.e., 1) and "$\preceq$" with the lowest reward (i.e., 0.25). Since the operator "#" does not have any refinement relation with other operators, we set it with the highest reward 1.

The second row in Table III shows the reward settings for relation clock holes. We design the reward mechanism for this kind of holes based on transitivity of the three relation operators (i.e., precedence, causality and subclock). For instance, assume that there is an incomplete CCSL specification

TABLE III
REWARD SETTINGS FOR HOLES

Hole Type	Side	Coincidence =	Exclusion #	SubClk$\subseteq$	Precedence $\prec$	Causality $\preccurlyeq$
Relation Operator	-	1	1	0.75	0.5	0.25
Relation Clock	**Left**	1	$\frac{sort(c_i)}{\|C\|}$	$\frac{sort(c_i)}{\|C\|}$	1-$\frac{sort(c_i)}{\|C\|}$	1-$\frac{sort(c_i)}{\|C\|}$
	Right	1	$\frac{sort(c_i)}{\|C\|}$	1-$\frac{sort(c_i)}{\|C\|}$	$\frac{sort(c_i)}{\|C\|}$	$\frac{sort(c_i)}{\|C\|}$
Expression Operator	**Left**	rank(e)	rank(e)	rank(e)	1-rank(e)	1-rank(e)
	Right	rank(e)	rank(e)	1-rank(e)	rank(e)	rank(e)
Expression Clock	**Left**	1	$\frac{sort(e)}{\|C\|}$	$\frac{sort(e)}{\|C\|}$	1-$\frac{sort(e)}{\|C\|}$	1-$\frac{sort(e)}{\|C\|}$
	Right	1	$\frac{sort(e)}{\|C\|}$	1-$\frac{sort(e)}{\|C\|}$	$\frac{sort(e)}{\|C\|}$	$\frac{sort(e)}{\|C\|}$

consisting of two constraints, i.e., $c_0 \prec \square_c^b$ and $c_1 \prec c_2$. We further assume that filling the hole with c_1 or c_2 will not violate the provided traces. In this case, we prefer using c_1 to fill the hole, since we can deduce constraints $c_0 \prec c_2$ and $c_1 \prec c_2$ from constraints $c_0 \prec c_1$ and $c_1 \prec c_2$, but not vice versa. To fulfill the above idea, we sort the clocks in ascending order by the number of ticks, and use $sort(c_i)$ to denote the ranking of the number of ticks for clock c_i. Let C denote the set of all the clocks and $||C||$ be the total number of clocks. For a clock hole in some relation constraint of type precedence, causality or subclock, we calculate its reward based on the tick count difference between the filled clock and its clock counterpart in the constraint. The smaller the difference is, the bigger the reward we can achieve. Note that for the three relation operators, when clock holes are located on different sides of relations, their ways of reward calculation is different. Unlike the above three relation operations, for the coincidence operator we set the rewards of filled clocks with the highest reward, i.e., 1. For the exclusion operator, we set one filled clock with more ticks with a bigger reward. This is because a clock with fewer ticks has a lower chance to exhibit some exclusion relation with other clocks.

The third row in Table III defines the reward settings of associated relation operators that affect the operator generation for a possible expression clock. By analyzing the characteristics of expression operators, we find that an expression operator hole "$e = c_1 \square_o^d c_2$" may have four possible selections, i.e., $c_1 + c_2$, $c_1 * c_2$, $c_1 \wedge c_2$ and $c_1 \vee c_2$, where $(c_1 + c_2) \preccurlyeq (c_1 \vee c_2) \preccurlyeq (c_1 \wedge c_2) \preccurlyeq (c_1 * c_2)$. As shown in Table IV, we use $rank(e)$ to indicate the ranking of the expression operators based on the number of ticks of the generated expression clock (i.e., e). If e is used in multiple relations, we need to calculate the reward for each of these relations, and use their average value as the reward of filling $\square_o^d$.

TABLE IV
RANK VALUES FOR EXPRESSION OPERATORS

Operator	Intersection $*$	Supermum $\wedge$	Infimum $\vee$	Union $+$
rank(e)	0	0.33	0.66	1.0

Similar to expression operator holes, the reward of an expression clock hole is determined by its associated relations. Since different selection leads to a different number of ticks of generated expression clock, we evaluate the reward of expression clock holes and relation clock holes in the same way as shown in the last row of Table III. For holes involving multiple relations, we calculate the reward of each involved relation and use the average value as the reward.

IV. PERFORMANCE EVALUATION

We evaluated the performance of our RL-based approach by conducting extensive experiments on a set of CCSL specifications. They are collected from four sources: i) eleven benchmarks provided by the CCSL simulation tool *TimeSquare* [23] that covers all the operators defined in Table I; ii) three benchmarks provided by the state-of-the-art CCSL synthesis tool *CCSLSketch* [27], which are listed in [30]; iii) one synthetic CCSL specification that was generated manually by ourselves; and iv) one complex CCSL specification for an industrial land gear system [39], which models the timing behaviors of an aircraft undercarriage.

In our experiments, we considered all the collected CCSL specifications without holes as the golden reference specifications for target systems. To enable the synthesis, firstly we used the CCSL specification simulation tool *TimeSquare* [23] to generated timing traces of a given length for each specification. Then, for each CCSL specification, we obtained its incomplete version by randomly removing clocks and operators from its relation and expression constraints. Note that for each relation or expression constraint, we replaced only one clock or one operator with a hole. Based on our proposed RL-based method, we implemented our CCSL synthesizer using Python. All the experimental results were generated on a laptop with Intel i7 2.5GHz CPU and 16GB memory.

A. Performance Analysis

Table V presents the synthesis results for all the collected CCSL specifications. The first two columns show the sources and indices of the CCSL specifications, respectively. The third column has three sub-columns, which show the number of clocks, expressions, and relations for a specific CCSL specification, respectively. The fourth column shows the hole settings (i.e., the number of clock holes, expression operator holes, and relation operator holes, respectively) in its three sub-columns. Due to space limitations, for each specification except the ones collected from [23], we only present the results of two different sets of incomplete constraints. Note that

TABLE V
Synthesis Results for CCSLSketch Benchmarks and Synthetic CCSL Specifications

Source	Index	CCSL Statistics			Hole Settings			CCSLSketch	# of	Our Approach	Speedup	Accuracy
		Clock	Expr.	Relation	Clock	Exp. Op.	Rel. Op.	Time (ms)	Rounds	Time (ms)		(%)
TimeSquare [23]	$S_=$	2	0	1	0	0	1	867	200	83	10.45	100
	$S_<$	2	0	1	0	0	1	787	200	74	10.64	100
	$S_\leq$	2	0	1	0	0	1	754	200	66	11.42	100
	$S_\subseteq$	2	0	1	0	0	1	791	200	60	13.18	100
	$S_\#$	2	0	1	0	0	1	685	200	65	10.54	100
	S_*	3	1	1	0	1	0	6639	200	149	44.56	100
	S_+	3	1	1	0	1	0	6787	200	129	52.61	100
	$S_\vee$	3	1	1	0	1	0	6922	200	145	47.74	100
	$S_\wedge$	3	1	1	0	1	0	6880	200	143	48.11	100
	$S_\$$	2	1	1	0	1	0	886	200	79	11.2	100
	$S_\propto$	2	1	1	0	1	0	908	200	61	14.89	100
CCSLSketch [30]	$Spec_1$	4	1	3	0	0	3	13782	2000	1159	11.89	100
									5000	2756	5.00	100
					0	1	2	36716	2000	1106	33.20	100
									5000	2740	13.40	100
	$Spec_2$	10	5	10	0	0	10	27689	2000	2725	10.16	100
									5000	6709	4.12	100
					4	2	4	40872	2000	3383	12.08	99.6
									5000	8450	4.83	99.8
	$Spec_3$	20	6	16	9	1	7	433523	2000	5758	75.29	100
									5000	14457	29.99	100
					5	1	10	42166	2000	5458	7.73	100
									5000	13550	3.11	100
Synthetic	$Spec_4$	9	4	5	0	4	0	NA	2000	4093	NA	100
									5000	10198	NA	100
					5	1	1	NA	2000	2984	NA	84.6
									5000	7612	NA	99.6
LandGear [39]	$Spec_5$	25	74	54	1	1	5	NA	2000	19376	NA	96.8
									5000	51306	NA	100
					3	1	0	NA	2000	31716	NA	99.8
									5000	80644	NA	99.6

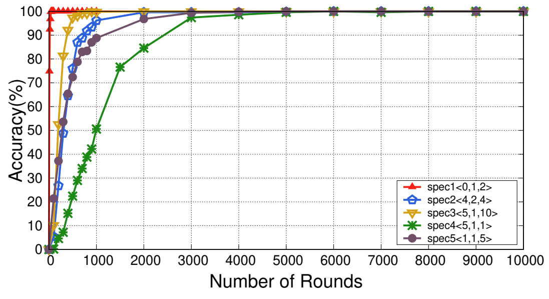
for each specification we generated five timing traces using *TimeSquare* to guide the synthesis, where each behavior has a length of 50. The fifth column shows the *CCSLSketch* time. In this column, we use *"NA"* to indicate that the results cannot be obtained due to the state explosion problem, which results in the crash of *CCSLSketch*.

The last four columns of Table V show the results using our proposed RL-based method. The sixth column indicates the number of rounds of reinforcement learning, and the seventh column shows the corresponding synthesis time for the given number of rounds. The eighth column presents the speedup of our approach against *CCSLSketch*. Note that, due to the randomness in RL, an incomplete CCSL specification together with the same experimental settings may lead to different complete CCSL specifications by using our approach. To fairly evaluate the accuracy of the generated CCSL specifications, for each incomplete specification we ran our approach 500 times and obtained 500 synthesized specifications. We compared the syntax between the 500 synthesized specifications and their golden reference counterpart. The last column shows the accuracy, which denotes the ratio of the synthesized specifications that have the same syntax as the golden reference specification. Note that in the experiment all the synthesized specifications with different syntax can satisfy their given five timing traces. Here, an inaccurate synthesized specification does not mean an incorrect one. Rather, it means that the synthesized specification is not as "tight" as its golden reference counterpart.

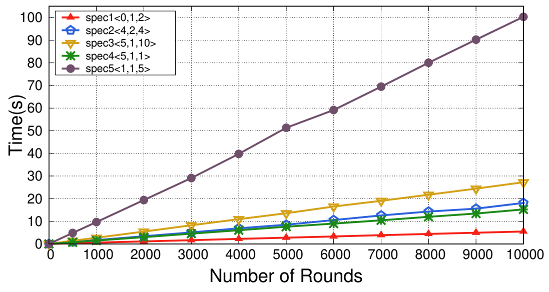
From Table V it can be observed that our approach outperforms the *CCSLSketch* for all the cases. It significantly reduces the synthesis time by up to 75.29X. Meanwhile, we can find that *CCSLSketch* failed to synthesize the four incomplete specifications (i.e., the two of $Spec_4$ and the two of $Spec_5$), while our approach can quickly figure out the synthesis results. This is because both of $Spec_4$ and $Spec_5$ have interrelated incomplete constraints that cannot be handled by *CCSLSketch*, such as "$e_0 \prec e_2$, $e_0 = c_0?c_1$, $e_2 = c_3?c_4$" in the second case of $Spec_4$. Note that the solution space size of the second case of $Spec_4$ (with 5 clock holes, 1 expression operator hole, and 1 relation operator hole) is $(9+4)^5 \times 4 \times 8 = 11,881,376$. However, our approach only uses 5000 rounds to figure out the tightest solution. In other words, our method can achieve an accuracy of 99.6% by only exploring 0.042% solution space, costing only 7.6 seconds.

B. Scalability Analysis

This sub-section tries to figure out which factors strongly affect the scalability of our method in terms of synthesis accuracy and time. Since our approach is based on RL where the number of training rounds plays an important role, we firstly investigated the impacts of the number of rounds on the synthesis results. Figure 5 shows the accuracy and time information for different number of rounds. In this figure, we only considered one incomplete version for each CCSL specification. We use the notation $Spec\langle x,y,z\rangle$ to denote a specific case shown in Table V, where *Spec* indicates the specification index and x, y and z denote the hole settings for clocks, expression operators and relation operators, respectively. Note that here for each incomplete specification, we generated five timing traces (each with a length of 50) to guide the synthesis.



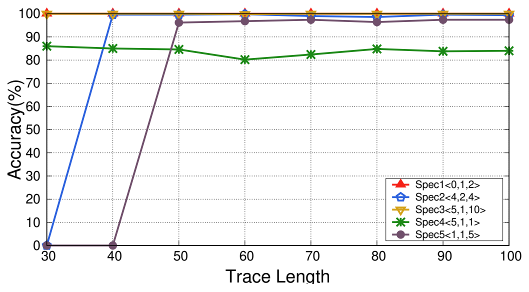
(a) Trend of accuracy



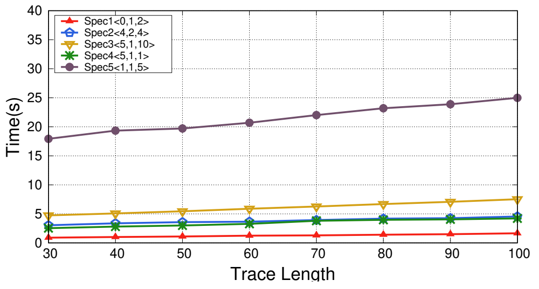
(b) Trend of time

Fig. 5. Impacts of the number of rounds

From Figure 5(a), we can find that when the number of RL rounds is larger than 3000, our approach can achieve an accuracy of at least 97.4% for all the five cases. When the number of rounds is larger than 5000, the least accuracy of the five cases increases up to 99.6%. By analyzing the constraints of five incomplete CCSL specifications, we found that when the number of rounds is fixed, a less number of correlated holes in some specification will lead to higher accuracy of the synthesis result. Moreover, as shown in Figure 5(a), the number of rounds plays an important role in determining the synthesis accuracy. Generally, a larger number of RL rounds for specification synthesis indicates that more solution space has been explored, thus the chance of finding the golden solution becomes higher. From Figure 5(b), we can observe that the synthesis time is linearly increased along with the increase of the number of rounds.



(a) Trend of accuracy

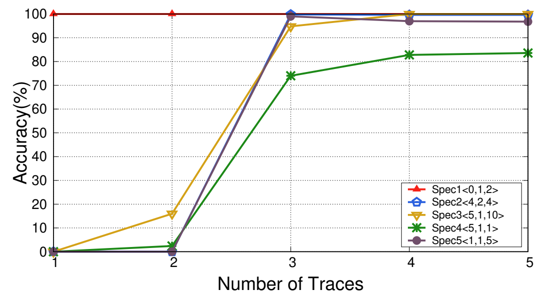


(b) Trend of time

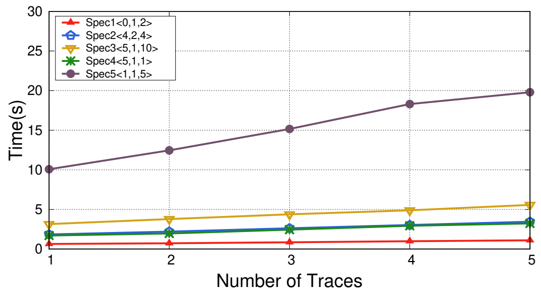
Fig. 6. Impacts of trace length

Figure 6 illustrates the impacts of trace lengths on synthesis accuracy and time. For the synthesis of each incomplete specification, we set the number of rounds to 2000, and each synthesis process is guided by five timing traces. From Figure 6(a), we can find that when the lengths of timing traces are 30, only three of the five cases can achieve their highest accuracy. When the lengths of timing traces increase to 80, we can observe the convergence of synthesis, where all of the five cases achieve their highest accuracy. It means that, for an incomplete specification, the longer traces we can obtain, the better synthesis accuracy we can achieve. This is mainly because short traces only present limited behaviors of a target system (e.g., some clocks may not tick in all the given traces), which makes the evaluator difficult to figure out the reward of multiple solutions. As a result, the specification synthesized based on short traces may not be the golden one. Therefore, we suggest to use longer traces to ensure the accuracy of our proposed synthesis method. Figure 6(b) shows the corresponding synthesis time for CCSL specifications with different lengths of provided timing traces. Along with the increase of lengths of timing traces, we can observe a linear trend for synthesis time.

Figure 7 shows the impacts of the number of timing traces. In this experiment, we set the lengths of timing traces to 50, and each synthesis took 2000 rounds. From Figure 6(a), we can find that when more timing traces are provided, we can achieve better accuracy for specification synthesis. In this experiment, when we use more than four timing traces, our approach can almost achieve the highest accuracy. Generally, when more traces are involved in the synthesis, the higher accuracy we can achieve, since more state transition information is provided to help the constraint construction. Figure 6(b) shows that the overall synthesis time is linear to the number of traces.



(a) Trend of accuracy



(b) Trend of time

Fig. 7. Impacts of the number of traces

V. Related Work

Although CCSL-based methods have been increasingly used in the design of real-time embedded systems, most of them focused on the formalization and verification of CCSL specifications. For example, Mallet and Simone [22] established correctness properties on MARTE/CCSL specifications by using state-based semantics for CCSL. Yin *et al.* [40] and Zhang *et al.* [41], [42] proposed various effective approaches that enable the schedulability analysis of CCSL specifications. Peters *et al.* [21] verified clock constraints against the given instant relations by converting CCSL constraints into automata. Zhang and Ying [24] presented an SMT-based approach to model checking the temporal properties specified for CCSL. However, most existing methods are based on the assumption that all the given CCSL specifications are complete, which is not always true in practice.

As a promising design automation method, specification synthesis attracts more and more attentions since it supports automated generation of formal specifications from requirements, system execution logs, or other textual descriptions. To enable the synthesis of CCSL specifications from their incomplete counterparts, Hu *et al.* [27] proposed CCSLSketch, which can encode incomplete CCSL constraints into sketching problems. However, since sketching strongly relies on the underlying SMT solving, the efficiency of CCSLSketch is inevitably restricted. When more holes are involved in the synthesis of CCSL specifications, CCSLSketch requires much more time and resources to achieve the synthesis results. To quickly figure out complete specifications, more and more specification synthesis methods resorted to AI techniques such as reinforcement learning, since they are good at enumerating candidate solutions. Meanwhile, to further improve the synthesis performance, deduction operations can be applied during the enumeration to effectively prune the unfruitful search spaces. Therefore, by combining both RL-based enumeration and proper deduction, the synthesis process would become more efficient and precise. For example, $\lambda 2$ [43] and Flesh-Meta [44] used inverse semantics or refutation to decompose the synthesis task and guide the program search. Chen *et al.* [28] developed a domain-specific language synthesis algorithm by combining both deductive and statistical reasoning mechanisms. The algorithm used reinforcement learning to search for implementations and adopts SMT-based deduction engine to guide the search. Huang *et al.* [29] introduced a cooperative synthesis framework named DryadSynth, which splits a synthesis problem into subproblems and solves them by using both enumeration and deduction operations. Inspired by the above RL-based methods, our method can deal with more complex CCSL specifications than CCSLSketch. As shown in the experiment section, our approach can quickly fill the holes for incomplete CCSL specifications, while CCSLSketch often crashes halfway. To the best of our knowledge, our work is the first attempt to apply both RL-based enumeration and deduction techniques for CCSL specification synthesis.

VI. Conclusion

Automated synthesis of Clock Constraint Specification Language (CCSL) has become popular in timing behavior modeling of real-time embedded systems. It can be used to improve designers' productivity as well as ensure the correctness and accuracy of target designs. However, Due to the lack of expertise in formal modeling and efficient synthesis tools, it is hard for requirement engineers to quickly and accurately figure out CCSL specifications from natural language-based design descriptions. To address this problem, in this paper we proposed an efficient Reinforcement Learning (RL)-based method for the co-synthesis of CCSL specifications based on our proposed state representations and reward mechanisms. Since the complexity of hole enumeration is exponential, we introduced various deduction-based optimization methods, which can not only prune unfruitful search space, but also enable the priority-based evaluation to guide the search process. Based on the efficient deduction-guided enumeration with RL, the overall CCSL specification synthesis time can be drastically reduced. Comprehensive experimental results on both well-known benchmarks and synthetic examples showed both the efficiency and the scalability of our approach.

Acknowledgments

This work was supported by the National Key Research and Development Program of China 2018YFB2101300, Natural Science Foundation of China 61872147 and 61872146, and ECNU Academic Innovation Promotion Program for Excellent Doctoral Students YBNLTS2020-041. Mingsong Chen is the corresponding author (mschen@sei.ecnu.edu.cn).

References

[1] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio, “A model-driven design environment for embedded systems,” in *Proceedings of Design Automation Conference (DAC)*, 2006, pp. 915–918.

[2] M. Chen, X. Qin, H. Koo, and P. Mishra, “System-Level Validation: High-Level Modeling and Directed Test Generation Techniques,” Springer, 2013.

[3] E. Letier and A. Van Lamsweerde, “Deriving operational software specifications from system goals,” *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 6, pp. 119–128, 2002.

[4] J. Sifakis, “Modeling real-time systems-challenges and work directions,” in *Proceedings of International Workshop on Embedded Software*, 2001, pp. 373–389.

[5] F. Thoen and F. Catthoor, “Modeling, verification and exploration of task-level concurrency in real-time embedded systems,” Springer Science & Business Media, 2012.

[6] A. V. Lamsweerde and E. Letier, “Handling obstacles in goal-oriented requirements engineering,” *IEEE Transactions on Software Engineering (TSE)*, vol. 26, no. 10, pp. 978–1005, 2000.

[7] Object Management Group, “UML profile for MARTE: Modeling and analysis of real-time embedded systems,” 2011.

[8] J. Vidal, F. De Lamotte, G. Gogniat, P. Soulard, and J.P. Diguet, “A co-design approach for embedded system modeling and code generation with UML and MARTE,” in *Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2009, pp. 226–231.

[9] M. Z. Iqbal, S. Ali, T. Yue, and L. Briand, “Experiences of applying UML/MARTE on three industrial projects,” in *Proceedings of International Conference on Model Driven Engineering Languages and Systems*, 2012, pp. 642–658.

[10] A. Koudri, D. Aulagnier, D. Vojtisek, P. Soulard, C. Moy, J. Champeau, J. Vidal, and J.C. Le Lann, “Using MARTE in a co-design methodology,” in *Proceedings of UML Workshop at DATE*, 2008.

[11] C. André and F. Mallet, “Specification and verification of time requirements with CCSL and Esterel,” in *Proceedings of ACM conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2009, pp. 167–176.

[12] J. Peters, R. Wille, N. Przigoda, U. Kühne, and R. Drechsler, “A generic representation of ccsl time constraints for UML/MARTE models,” in *Proceedings of Design Automation Conference (DAC)*, 2015, pp. 1–6.

[13] M. Zhang, F. Dai, and F. Mallet, “Periodic scheduling for marte/ccsl: Theory and practice,” *Science of Computer Programming*, vol. 154, pp. 42–60, 2018.

[14] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communication of ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[15] F. Mallet, “Automatic generation of observers from MARTE/CCSL,” in *Proceedings of International Symposium on Rapid System Prototyping (RSP)*, 2012, pp. 86–92.

[16] C. André, F. Mallet, and M.A. Peraldi-Frati, “Multiform time in UML for real-time embedded applications,” in *Proceedings of International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2007, pp. 232–237.

[17] R. Drechsler, M. Soeken, and R. Wille, “Formal specification level: Towards verification-driven design based on natural language processing,” in *Proceeding of Forum on Specification and Design Languages (FDL)*, 2012, pp. 53–58.

[18] F. Gao, F. Mallet, M. Zhang, and M. Chen, “Modeling and verifying uncertainty-aware timing behaviors using parametric logical time constraint,” in *Proc. of Design, Automation and Test in Europe Conference (DATE)*, 2020, pp. 376-381.

[19] M. Zhang, F. Dai, and F. Mallet, “Periodic scheduling for MARTE/CCSL: Theory and practice,” *Science of Computer Programming*, vol. 154, pp. 42–60, 2018.

[20] F. Mallet and M. Zhang, “Work-in-Progress: From logical time scheduling to real-time scheduling,” in *Proceedings of Real-Time Systems Symposium (RTSS)*, 2018, pp. 143–146.

[21] J. Peters, N. Przigoda, R. Wille, and R. Drechsler, “Clocks vs. instants relations: Verifying CCSL time constraints in UML/MARTE models,” in *Proceedings of International Conference on Formal Methods and Models for System Design (MEMOCODE)*, 2016, pp. 78–84.

[22] F. Mallet and R. De Simone, “Correctness issues on MARTE/CCSL constraints,” *Science of Computer Programming*, vol. 106, pp. 78–92, 2015.

[23] J. DeAntoni and F. Mallet, “Timesquare: Treat your models with logical time,” in *Proceedings of International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, 2012, pp. 34–41.

[24] M. Zhang and Y. Ying, “Towards SMT-based LTL model checking of clock constraint specification language for real-time and embedded systems,” in *Proceedings of ACM Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2017, pp. 61–70.

[25] R. Drechsler, M. Soeken, and R. Wille, “Formal specification level: Towards verification-driven design based on natural language processing,” in *Proceeding of Forum on Specification and Design Languages*, 2012, pp. 53–58.

[26] F. Balarin and R. Passerone, “Specification, synthesis, and simulation of transactor processes,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 26, no. 10, pp. 1749–1762, 2007.

[27] M. Hu, T. Wei, M. Zhang, F. Mallet, and M. Chen, “Sample-guided automated synthesis for CCSL specifications,” in *Proceedings of Design Automation Conference (DAC)*, 2019, pp. 1–6.

[28] Y. Chen, C. Wang, O. Bastani, I. Dillig, and Y. Feng, “Program synthesis using deduction-guided reinforcement learning,” in *Proceedings of International Conference on Computer Aided Verification (CAV)*, 2020, pp. 587–610.

[29] K. Huang, X. Qiu, P. Shen, and Y. Wang, “Reconciling enumerative and deductive program synthesis,” in *Proceedings of ACM Conference on Programming Language Design and Implementation (PLDI)*, 2020, pp. 1159–1174.

[30] CCSLSketch, https://github.com/HMHelloWorld/CCSLSketch.

[31] C. André, F. Mallet, and M.A. Peraldi-Frati, “A multiform approach to real-time system modeling: Application to an automotive system,” in *Proceedings of International Symposium on Industrial Embedded Systems (SIES)*, 2007, pp. 234–241.

[32] M. Zhang AND F. Song and F. Mallet and X. Chen, “SMT-Based Bounded Schedulability Analysis of the Clock Constraint Specification Language,” in the 22nd Fundamental Approaches to Software Engineering (FASE), ETAPS 2019, pp. 61-78, Springer.

[33] D. Yue, V. Joloboff, and F. Mallet, “Flexible runtime verification based on logical clock constraints,” in *Proceedings of Forum on Specification and Design Languages (FDL)*, 2016, pp. 1–8.

[34] R. Bavishi, C. Lemieux, R. Fox, K. Sen, and I. Stoica, “AutoPandas: neural-backed generators for program synthesis,” *Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2019, pp. 168:1–168:27.

[35] V. Kurin, S. Godil, S. Whiteson, and B. Catanzaro, “Improving SAT solver heuristics with graph networks and reinforcement learning,” *arXiv preprint arXiv:1909.11830*, 2019.

[36] J. Marques-Silva, I. Lynce, and S. Malik, “Conflict-driven clause learning sat solvers,” in *Handbook of satisfiability*, 2021, pp. 133–182.

[37] D. Lee, H. Seo, and M. W. Jung, “Neural basis of reinforcement learning and decision making,” *Annual Review of Neuroscience*, vol. 35, pp. 287–308, 2012.

[38] M. Van Otterlo and M. Wiering, “Reinforcement learning and Markov decision processes,” *Reinforcement Learning*, pp. 3–42, 2012.

[39] F. Boniol and V. Wiels, “The landing gear system case study,” in *Proceedings of International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ)*, 2014, pp. 1–18.

[40] L. Yin, J. Liu, Z. Ding, F. Mallet, and R. De Simone, “Schedulability analysis with CCSL specifications,” in *Proceedings of Asia-Pacific Software Engineering Conference (APSEC)*, 2013, pp. 414–421.

[41] Y. Zhang, F. Mallet, H. Zhu, and Y. Chen, “A logical approach for the schedulability analysis of CCSL,” in *Proceedings of International Symposium on Theoretical Aspects of Software Engineering (TASE)*, 2019, pp. 25–32.

[42] Y. Zhang, F. Mallet, H. Zhu, Y. Chen, B. Liu, and Z. Liu, “A clock-based dynamic logic for schedulability analysis of CCSL specifications,” *Science of Computer Programming*, vol. 202, p. 102546, 2021.

[43] J. K. Feser, S. Chaudhuri, and I. Dillig, “Synthesizing data structure transformations from input-output examples,” in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015, pp. 229–239.

[44] O. Polozov and S. Gulwani, “Flashmeta: A framework for inductive program synthesis,” in *Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015, pp. 107–126.